

## Contents

<b>1</b>	<b>Forelesning 1</b>	<b>2</b>
1.1	Hva er en Algoritme? . . . . .	2
1.2	Forelesninger . . . . .	2
1.3	Pensum . . . . .	2
1.4	Beregne summen fra 1 til $n$ . . . . .	2
1.5	Hvordan forenkle beregninger? . . . . .	2
<b>2</b>	<b>Forelesning 2</b>	<b>3</b>
2.1	Hva er en datastruktur . . . . .	3
2.2	Array List . . . . .	3
2.3	Linked List . . . . .	3
2.4	Hvordan finne kjøretid på metoder? . . . . .	3
2.5	Kø og Stabel . . . . .	4
2.5.1	Metoder i Queue og Stack . . . . .	4
2.6	Set . . . . .	4
<b>3</b>	<b>Forelesning 3</b>	<b>4</b>
3.1	Big $O$ Kjøretid . . . . .	4
3.2	Doubling Rate . . . . .	5
<b>4</b>	<b>Forelesning 4</b>	<b>5</b>
4.1	Rekursjon - Fibonacci . . . . .	5
4.2	Binære trær . . . . .	5
4.3	Binærsøk . . . . .	5
4.4	Sortering - hvorfor? . . . . .	5
4.4.1	Hva er sortering? . . . . .	5
4.5	Sortering i Java . . . . .	6
4.5.1	Comparable Interface . . . . .	6
4.5.2	Sortere på forskjellige måter . . . . .	6
4.5.3	Mange sorteringsalgoritmer . . . . .	6
<b>5</b>	<b>Forelesning 5</b>	<b>7</b>
5.1	Selection Sort . . . . .	7
5.2	"In-Place" Sortering . . . . .	8
5.3	Insertion Sort . . . . .	8
5.4	Quick Sort . . . . .	8
<b>6</b>	<b>Forelesning 6</b>	<b>9</b>
6.1	Divide & Conquer . . . . .	9
6.2	Mergesort . . . . .	9
6.3	Nedre grense på kjøretid . . . . .	9
6.4	Sorter $n$ heltall . . . . .	10
6.5	Bucket Sort . . . . .	10

# 1 Forelesning 1

## 1.1 Hva er en Algoritme?

En algoritme er en plan for å løse et problem, ikke helt det samme som en implementasjon. Den tar en input og gir en output (metode i *Java*).

*Dagens Problem:* Gitt en liste med brukernavn, sjekk at ingen brukernavn er like.

## 1.2 Forelesninger

Forelesningene blir tatt opp og gjort tilgjengelig på Mitt UiB, og kode blir lagt ut på git.

## 1.3 Pensum

Algorithms by Robert Sedgewick 4<sup>th</sup> edition. Kapittel 1-4 er pensum.

1. Fundamentale Algoritmer og datastrukturer
2. Sortering
3. Søking
4. Grafer

## 1.4 Beregne summen fra 1 til n

Følgende formel brukes for å regne summen fra 1 til  $n$ .

$$1 + 2 + 3 + \dots + n = \sum_i^n \frac{n \cdot (n + 1)}{2}$$

Kjøretiden til implementasjonen kan estimeres til

$$\frac{n \cdot (n + 1)}{2} \cdot 20 \approx 10n^2$$

$$10 \cdot n^2 = 10^9 \\ \approx 10000$$

Kjøretid på et tilsvarende program som bruker `Collections.sort()` vil ha en kjøretid på ca.

$$10 \cdot n \log n + 10 \cdot n = 10^9$$

Når  $n$  blir stor, er alltid  $n \log n$  raskere enn  $n^2$ .

## 1.5 Hvordan forenkle beregninger?

Det er tidkrevende å kjøre programmet og ta tiden. Det er vanskelig å vite nøyaktig hvor mange operasjoner det er, derfor velger vi å beholde største ledd, og vi ser bort ifra konstanter. Som eksempel kan følgende formel forenkles.

$$\frac{n \cdot (n + 1)}{2} \cdot 20 \approx 10n^2 = O(n^2)$$

## 2 Forelesning 2

### 2.1 Hva er en datastruktur

En algoritme tar input, gjør beregninger, og gir output. Datastrukturer lagrer data og har flere forskjellige oppgaver som kan utføres på disse dataene.

#### Example. GPS Navigasjonsenhet

- Har veinettverk
- Kan be om korteste vei fra start til mål

Er dette en algoritme eller en datastruktur?



`Collection <E> interface` er en samling med objekter av typen `E`. Viktige metoder er

- `size()`
- `contains(Object o)`
- `add (E e)`
- `Remove(Object obj)`
- `toArray()`

`List` er en utvidelse av `Collection` med litt flere metoder. Elementene i en liste har en indeks, og noen viktige metoder i `List` er

- `indexOf(Object obj)`
- `get(int index)`
- `set(int index, E e)`

### 2.2 Array List

I `ArrayList` brukes en array av typen `Object[]`. Bare en del av arrayen har data. Når dette arrayet blir fullt må vi lage et større array. Dette betyr at det er lett å få `IndexOutOfBoundsException`. For eksempel kan man lage et nytt array av dobbel størrelse, hvor alle elementer fra forrige array kopieres over til det nye arrayet.

### 2.3 Linked List

En Linked List er en liste hvor hvert element i listen lagres i et eget node object. Hver node vet kun neste og forrige node. Listen vet kun første og siste node. For å finne node  $i$ , så starter vi på første node og "hopper"  $i$  ganger.

En linked list som kun vet neste element kalles en *Single Linked List*, og en linked list som vet både neste og forrige element kalles en *Double Linked List*.

### 2.4 Hvordan finne kjøretid på metoder?

Man må vite hvordan `ArrayList` og `LinkedList` er implementert. Ved å forstå hva `ArrayList` og `LinkedList` gjør, er det lett å forstå hva kjøretiden er. Ukesoppgaven er og implementere enkle versjoner av disse listene.

## 2.5 Kø og Stabel

Kø	Stabel
FIFO	FILO / LIFO
Legger til sist	Legger til sist
Fjerner først	Fjerner sist

### 2.5.1 Metoder i Queue og Stack

List	Queue	Stack
add(E e)	offer(E e)	push(E e)
remove(int i)	poll()	pop()
get(int i)	peek()	peek()

## 2.6 Set

Set er en Collection der hvert element kun kan være der en gang. Elementene har ikke en ebstemt ordning. Set har heller ikke `indexOf(element)` metoden.

	HashSet	TreeSet
add()	$O(1)^*$	$O(\log n)$
remove()	$O(1)$	$O(\log n)$
contains(obj)	$O(1)$	$O(\log n)$

## 3 Forelesning 3

### 3.1 Big O Kjøretid

Variablen  $n$  er vanligvis definert som størrelse på input, men den kan defineres til hva som helst. Når vi beregner kjøretid er  $f(n)$  max antall operasjoner PCen må gjøre når et program kjøres på en input av størrelse  $n$ . Det vil si, velg den verste input av størrelse  $n$ . Teoretisk er vi interessert i

$$\sum_{n=0}^{\infty} f(n)$$

Mens praktisk er vi interessert i når  $f(n)$  er  $10^9$  til  $10^{12}$ . Vi har at

$$\begin{aligned} 3 + 7 &\leq 2 \cdot 7 \\ 124 + 98 &\leq 2 \cdot 124 \end{aligned}$$

Dette gir oss følgende formel

$$a + b \leq 2 \cdot \max(a, b)$$

Som resulterer i

$$\begin{aligned}O(1) + O(1) &= O(1) \\O(1) + O(n) &= O(n) \\O(n) + O(n^2) &= O(n^2)\end{aligned}$$

Big O beskriver et sett med funksjoner.  $n^2 + 3n$  og  $7n^2$  er i  $O(n^2)$ . Alle funksjoner som er mindre er også med, dvs at  $3n$  og  $7n \log(n)$  er i  $O(n^2)$ .

**Definisjon 1.** En funksjon  $f(n)$  er i  $O(g(n))$  dersom det finnes konstanter  $c$  og  $N$  slik at  $f(n) < c \cdot g(n)$  for alle  $n > N$ , eller, skrevet rent matematisk

$$f(n) \in O(g(n)) \Rightarrow f(n) < c \cdot g(n) \forall n > N$$

### 3.2 Doubling Rate

Hva skjer med kjøretid når input blir dobbelt så stor?

- $O(1)$  - Ingenting, ikke avhengig av  $n$ .
- $O(n)$  -  $c \cdot n \rightarrow c \cdot 2n$ , dobbelt så lang tid.
- $O(n^2)$  -  $c \cdot n^2 \rightarrow c \cdot (2n)^2$ , fire ganger så stort.
- $O(n^3)$  -  $c \cdot n^3 \rightarrow c \cdot (2n)^3$ , 8 ganger så stort.

## 4 Forelesning 4

### 4.1 Rekursjon - Fibonacci

### 4.2 Binære trær

Et binært tres antall noder dobles for hvert nivå man går nedover treet.

### 4.3 Binærsøk

**Problem:** Sortert tabell med  $n$  element, leter etter et bestemt element. Hvor mange ganger må vi teste før vi er sikker på å finne det?

1. Forsøk -  $n$  element igjen: prøver midterste element
2. Forsøk -  $\frac{n}{2}$  element igjen: prøver igjen midterste element

Osv, til det kun er ett element igjen.

### 4.4 Sortering - hvorfor?

#### 4.4.1 Hva er sortering?

Sortering er en total ordning av en mengde  $A$ . F. eks følgende ordning

$$a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$$

Ordningen kan være anti-symmetrisk.

$$-a \leq b \wedge a \geq b \Rightarrow a = b$$

Eller Transitiv

$$a \leq b \wedge b \leq c \rightarrow a \leq c$$

Eller Total

$$a \leq b \vee b \leq a$$

## 4.5 Sortering i Java

Man kan sortere med f. eks `Collections.sort(List <T> list)`, men hvilken type må `T` være? Man må implementere `Comparable`. Med Java generics blir det litt mer komplisert, men man kan implementere `comparable` på følgende måte.

```
T extends Comparable <? super T>
```

### 4.5.1 Comparable Interface

- `public int compareTo(T obj);`
  - Returnerer -1 hvis `this < obj`
  - Returnerer 0 hvis `this == obj`
  - Returnerer 1 hvis `this > obj`
- Beskriver naturlig ordning
  - Integer, Double, String, Date...

### 4.5.2 Sortere på forskjellige måter

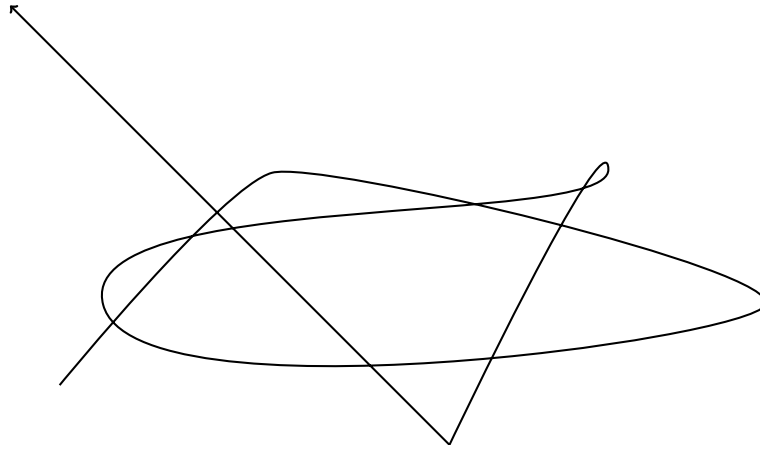
Da må man bruke `Comparator` interface

```
int compare(T o1, T o2)
```

### 4.5.3 Mange sorteringsalgoritmer

- Selection sort
- Merge Sort
- Heap Sort
- Bogo Sort (The GOAT)
- Insertion Sort
- Shell Sort
- Quick Sort
- Bucket Sort
- Radix Sort

- Bubble Sort



## 5 Forelesning 5

### 5.1 Selection Sort

- Finn det minste elementet
  - Flytt det først (index 0)
  - Finn minste elementet blant de  $n - 1$  siste.
  - Flytt det på andre posisjon (index 1)
- ... til alle elementene er sortert.

Hva er kjøretiden til denne funksjonen?

```

1  public <T extends Comparable<? super T>> void sort(List<T> list) {
2      List<T> sorted = new ArrayList<>();
3
4      while (!list.isEmpty()) { // n iterasjoner
5          T smallest = Collections.min(list); // O(n)
6          list.remove(smallest); // O(n)
7          sorted.add(smallest); // O(n)
8      }
9      list.addAll(sorted);
10 }

```

Figure 1: Selection sort har kjøretid  $O(n^2)$

## 5.2 "In-Place" Sortering

- Sortere uten å bruke ekstra minne
- Ikke lag ny liste, men bytt elementer i listen
- Dette forbedrer ikke big-O kjøretid, men kan redusere konstantene i kjøretiden
- Dette kalles optimering av kode

## 5.3 Insertion Sort

- Lag en ny liste
- Gå gjennom elementene

## 5.4 Quick Sort

- Finn et element kalt "pivot"
- Divide:
  - Første list  $\leq$  pivot
  - Siste list  $\geq$  pivot
  - Hva med de lik pivot?
- Sort hver liste rekursivt.
- Slå sammen disse listene
  - Siden vi delte inn i basert på pivot blir det
  - Første, pivot, siste
- Hva er kjøretiden til QuickSort
  - Det ligner litt på merge sort
  - Vi deler opp i to lister, men er de like store?



- Hvor mange ganger må vi dele opp?
- I verste tilfelle blir pivot det største eller minste elementet?
- I beste tilfelle blir pivot det midterste elementet.

Quicksort har altså runtime  $T \in (O(n \log(n)), O(n^2))$

## 6 Forelesning 6

### 6.1 Divide & Conquer

- En nyttig strategi når man skal finne effektive algoritmer.
- Divide
  - Del opp input og løs hver del for seg
- Conquer
  - Finn en måte å slå sammen de to delene til løsning for hele input

### 6.2 Mergesort

- Divide
  - Lag to lister som hver har halvparten av elementene
  - Sorter de to små listene
- Conquer
  - Gitt to små sorterte lister, slå disse sammen til en stor sortert liste.
- La oss implementere mergesort.

### 6.3 Nedre grense på kjøretid

- Vi kan se å sortering som å velge den rette ordningen blant alle mulige ordninger av en liste.
- Hvor mange måter er det å ordne  $n$  elementer? ( $n!$ )
- Tenk deg en liste av alle mulige ordninger
- Gjør en sammenligning av 2 elementer  $a$  og  $b$ .
- Blant alle ordninger i listen kan vi
  - Enten krysse ut de der  $a < b$
  - Eller krysse ut de der  $b < a$
- Hvor mange ganger må vi dele på 2 før  $n!$  blir 1?
- $\log(n!)$

### 6.4 Sorter $n$ heltall

- Hvor for kan vi sortere  $n$  heltall der alle tallene er
  - Mellom 0 og 1000
  - Mellom 0 og  $n$
  - Mellom 0 og  $n^2$

### 6.5 Bucket Sort

- Ikke egentlig en sorteringsalgoritme
- Deler opp i grupper (buckets/bins)
  - For alle par  $a, b$  har vi:  $a \leq b \leftrightarrow \text{bucket}(a) \leq \text{bucket}(b)$
- Trenger funksjon for bucket nummer
  - Eksempel: String, 26 buckets,  $a$  i første,  $b$  i andre.
- Radix sort ligner på bucket sort
  - Deler igjen opp hver bucket basert på 2. bokstav.